Cpr E 583 Adaptive Computing Systems	Fall Semester 1998
Lectures 15 & 16: October 13 & October 15, 1998	
Instructor: Arun Somani, Akhilesh Tyagi	Scribe: Srini Ramasubramanian

1 Introduction

In these two lectures, we look into the general procedure of how to map a given algorithm onto an array processor. The first question that arises is: *How should we represent the algorithm, so that the parallelism available in the algorithm is detected?* There is an inherent ordering present when the algorithms are represented. Hence, expressing an algorithm in an efficient way, so that the available parallelism can be detected is an important issue.

Algorithm expression is a basic tool for describing an algorithm for parallel and pipeline processing. This forms the basis for a systematic approach in mapping the algorithms to the array processors. An algorithm may be expressed in more than one way. In the next section, we look into some of the techniques that are presented in the literature for expressing parallel algorithms.

2 Parallel Algorithm Expressions

Parallel algorithm expressions may be derived using one of the following methods:

1. Vectorization of sequential algorithm expressions.
2. Direct parallel algorithm expressions.

2.1 Vectorization of Sequential Algorithm Expressions

Consider the matrix addition operation. Let A, B, and C be three matrices of the same order and let $C = A + B$. This can be written as:

$$C(i,j) = A(i,j) + B(i,j), \text{ for all i and j.}$$

The corresponding Fortran code is given by

```
      DO 10 J = 1,4
      DO 10 I = 1,4
      C(I,J) = A(I,J) + B(I,J)
10    CONTINUE
```

The above code is executed sequentially in FORTRAN. It can be seen that the matrices are accessed in the column major order. On many computers, the program will not be efficient, if this order is changed. This depends on how the vales of the matrices are stored in the memory. If no ordering is defined, then it is possible for the the compiler to choose the most efficient algorithm for a particular target computer, allowing some or all the parallelism available to be exploited. Since, algorithms are expressed as a sequential code, ordering becomes unavoidable. Hence, parallel expression of an algorithm is desirable.

A vector compiler compiler could translate the above FORTRAN code into the following prallel code. This code exploits parallelism along the i and the j direction. The first line in the code

```
IN PARALLEL FOR 1 ≤ I ≤ 4, 1 ≤ J ≤ 4 DO BEGIN
    C(I,J) = A(I,J) + B(I,J)
END IN PARALLEL DO
```

indicates that the computation specified in the loop can be done in parallel for all i and j values.

2.2 Direct Expression of Parallel Algorithms

Parallel algorithms can be expressed as snapshots, recursive algorithms with space-time indices, parallel codes, dependency graphs (DGs), or signal flow graphs. The concept of "Single Assignment Code" is used to a great extent in expressing parallel algorithms.

A **single assignment code** is a form of code where every variable is assigned exactly one value during the execution of the algorithm. For example, consider a matrix-vector multiplication given by $c = Ab$. The i th element of c can be written as,

$$c_i = \sum_{j=1}^{n} A_{ij} b_j .$$

The corresponding FORTRAN code is given by,

```
      DO 10 I = 1,4
      C(I) = 0
      DO 10 J = 1,4
      C(I) = C(I) + A(I,J) * B(J)
10    CONTINUE
```

Now, by using the single assignment rule, the same code can be rewritten as,
C which was initially a vector, has been made into a matrix during this process. It can be seen that the number of variables that are required in an algorithm is equal to the number of assignment operations that are performed in the algorithm. The result of the matrix-vector multiplication is stored in $C(I, 5)$, where $1 \leq I \leq 4$.

```
      DO 10 I = 1,4
      C(I,0) = 0
      DO 10 J = 1,4
      C(I, J+1) = C(I, J) + A(I,J) * B(J)
10    CONTINUE
```

2.2.1 Recursive Algorithms

Recursive algorithms are one of the most convenient and concise representation of an algorithm. The expressions are represented in terms of space-time indices. It can be seen that the recursive algorithms come under the category of single assignment code. For example, the matrix-vector multiplication given by, c=Ab, can be written as,

$$C_i^{(j+1)} = C_i^{(j)} + a_i^{(j)} b_i^{(j)}$$

where j denotes the recursion index, or equivalently a time step, i, j = 1, 2, ... , N, and

$$\begin{aligned} C_i^{(1)} &= 0 \\ a_i^{(j)} &= A(i,j) \\ b_i^{(j)} &= b(j) \end{aligned}$$

2.2.2 Snapshot

A snapshot is a description of an algorithm at a particular instant of time. Snapshot for matrix-vector multiplication on a linear array processor is shown in Figure 1. The figure shows the snapshot at time steps 0, 1, and 2.

2.2.3 Dependency Graph (DG)

The amount of parallelism that is available in an algorithm depends on the amount of data dependency present. A *Dependence Graph* is a graph that shows the dependency of the computations in an algorithm. It is a graphical representation of the single assignment algorithm.

Graph Terminologies: A graph $G(N, A)$ is a set N containing the nodes, and a set A containing the arcs, or edges, that connect these nodes. Each arc, $a \in A$, connects a pair of nodes, $i, j \in N$, and is represented as $a = i \xrightarrow{a} j$. i is called the *initial* node and the j is called the *terminal* node of the edge a.

A *loop* is an arc where the initial and the terminal nodes are the same. A chain is a sequence of distinct arcs, $C = \{a_1, a_2, ..., a_c\}$, such that an arc $a_q \in C$, where $1 < q < c$, has one of its node common with the arc a_{q+1} and the other common with a_{q-1}, without considering the direction of the arcs. If in a chain, all the arcs are directed the same way, it is called a *path*. If in the chain C, the initial node of the arc a_1 is same as the terminal node of the arc a_c, then, it is called an *undirected cycle*. If the same is true in a path, it is called a *cycle*. An *elementary* chain, path, undirected cycle and cycle is one in which no node is present more than once, except the starting

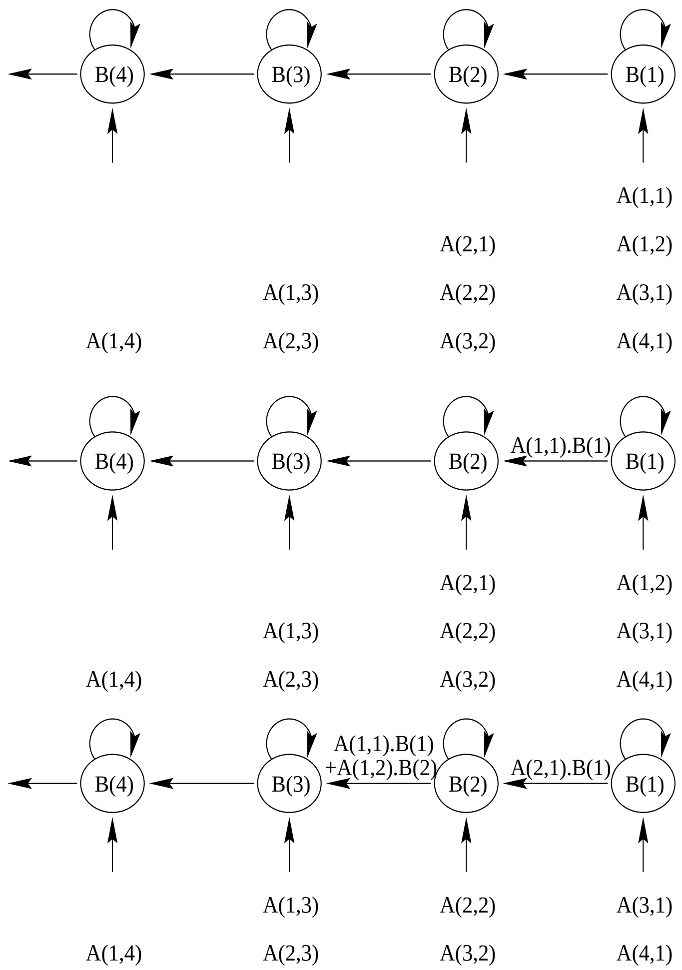

Figure 1: Snapshot of matrix-vector multiplication on array processor

and the ending nodes which *may* be the same. A graph is said to be connected if there exists a chain between every pair of nodes and is said to be strongly connected, if there exists a path between every ordered pair of nodes.

The dependency graph for the matrix-vector multiplication is shown in Figure 2. The internal computation of each node is ignored in the DG. A DG is said to be *complete* if the operations that are performed by every node is also included. An algorithm is *computable* iff the DG contains no loops or circles.

It can be seen from the above figure that the value of B has to be broadcasted. In other words, we need global communication. This could be replaced by communication that take place only between the adjacent nodes.

Localized Dependency Graph A localized dependence graph is one in which the communication takes place only between the neighboring nodes. The DG with local communication for

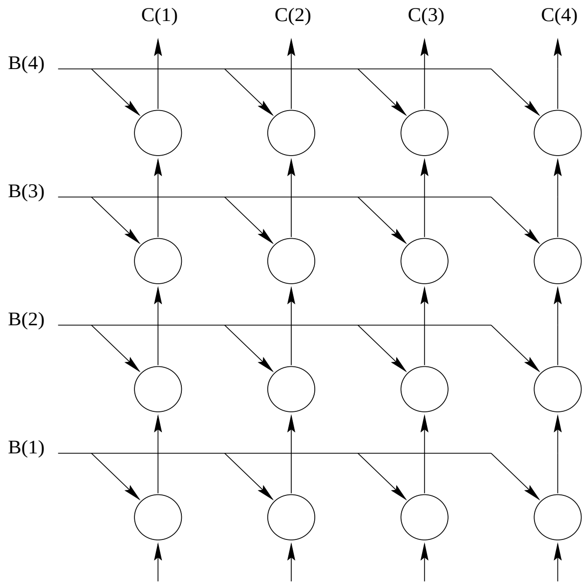

Figure 2: DG for matrix-vector multiplication with global communication

the matrix-vector multiplication is shown in Figure 3. Note that in the above example, we have a two-dimensional DG. It is possible to have an n-dimensional DG, where $n > 2$. From the dependency graph, we can see which nodes can start their computation at the same time. The set of nodes that start their computation at the same time are said to form the *equitemporal hyperplane*.

A DG is said to be *shift-invariant* if the dependence arcs corresponding to all the nodes in the index space are independent of their positions. In other words, if there is an incoming (outgoing) arc to (from) node i_1 from (to) a node at a distance j, then the same is true for any other node.

2.3 Examples

In this section, we take the examples of sorting and convolution and construct the DG. For convolution, we show the DG with both global and local communication.

2.3.1 Sorting

Given a set $A = \{a(i)\}$, where i=1, 2, ..., N. we need to sort the elements of A. Let $Y = \{y(i)\}$ represent the sorted set. The DG for sorting in the descending order, for $N = 4$, with local communication is shown in Figure 4.

Each node does the computation of finding out maximum and minimum values of the two input values. The function of each node can be written as,

$$\begin{aligned} m(i+1,j) &= max[x(i,j), m(i,j)] \\ x(i,j+1) &= min[x(i,j), m(i,j)] \end{aligned}$$

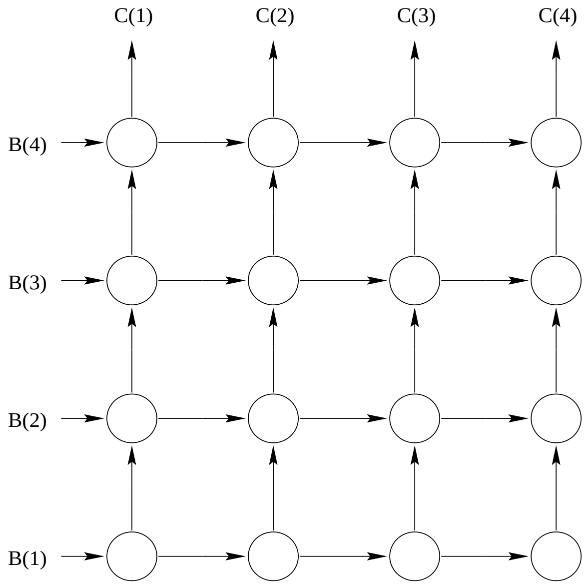

Figure 3: DG for matrix-vector multiplication with local communication

where $x(i,1) = a(i)$, $m(5,j) = y(j)$, $m(1,j) = -\infty$, and i, j = 1, 2, ..., N.

2.3.2 Convolution

Given two inputs $u(i)$ and $w(i)$, for $1 \leq i \leq N$, the convolution of u and w is given by the formula

$$y(j) = \sum_{k=0}^{N} u(k)w(j-k)$$

where, $1 < j < 2N-1$ and $u(i), w(i) = 0$ for $i <= 0$ or $i > N$. The DG, for $N = 4$, with the global communication is shown in Figure 5. This figure shows the DG when y is fed along two different directions. The DGs are then converted to have only local communication, shown in Figure 6.

3 Canonical Mapping

Given any algorithm, we can map it onto an array processor in three steps. They are:

1. **Design of DG**
   This involves indentifying a suitable algorithm and constructing a corresponding DG. This step is very critical because, the design of the array processor depends on a great extent on the algorithm that is chosen and the way the dependencies are dealt. The techniques discussed above could be used for this purpose.

2. **Design of SFG**
   The constructed DG is then converted into a signal flow graph. It is possible to obtain several SFG's for a given DG. This step defines the computations that need to be done at each node.

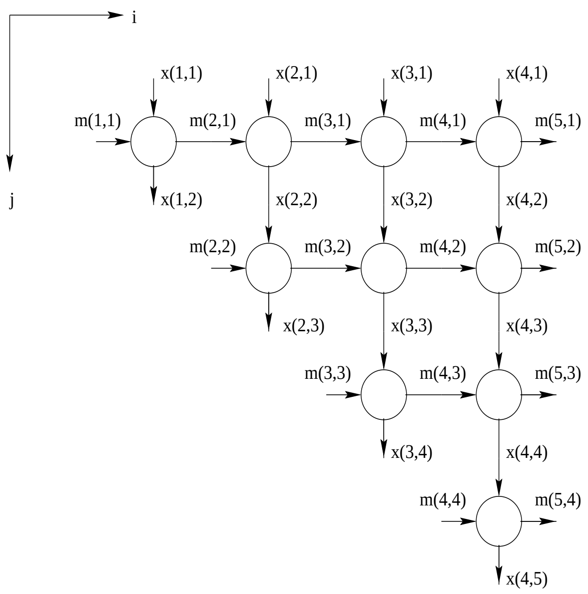

Figure 4: DG for sorting with local communication

3. **Design of Array Processor**
   The final step is to implement the above SFG using SIMD, MIMD, systolic, or wavefront processor.

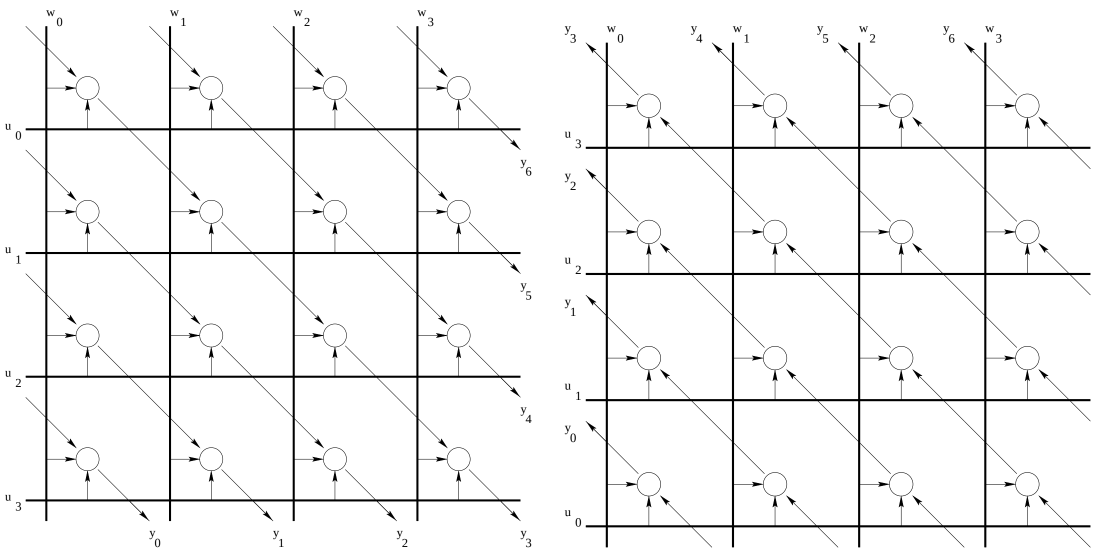

Figure 5: DG for convolution with the inputs broadcasted

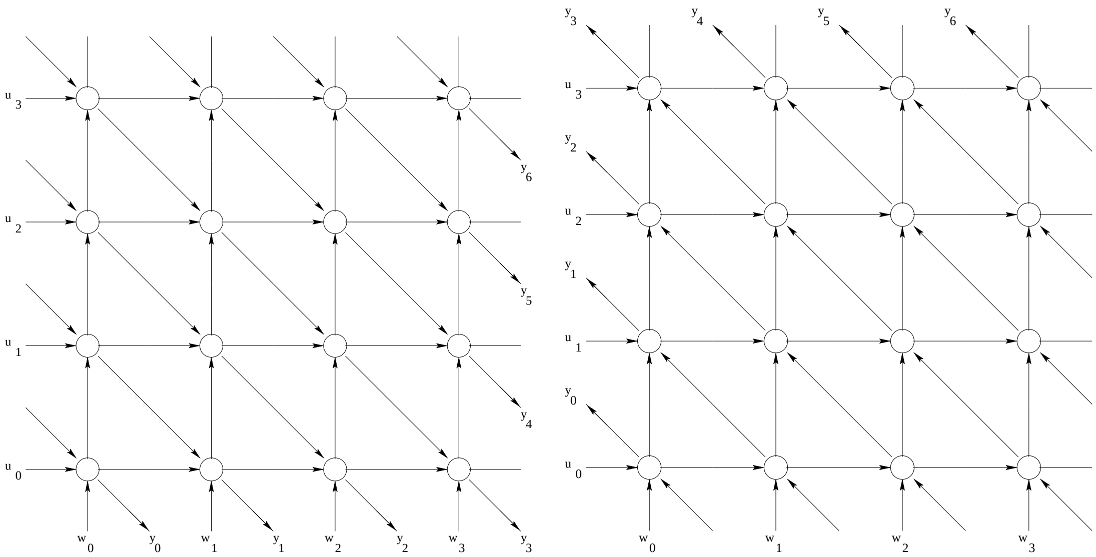

Figure 6: DG for convolution with local communication